Scalable Algorithms for Nearest-Neighbor Joins on Big Trajectory Data

Yixiang Fang, Reynold Cheng, *Member, IEEE,* Wenbin Tang, Silviu Maniu, and Xuan Yang

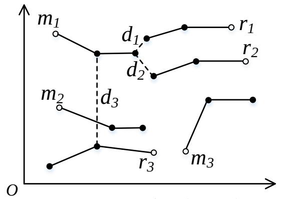
Abstract—Trajectory data are prevalent in systems that monitor the locations of moving objects. In a location-based service, for instance, the positions of vehicles are continuously monitored through GPS; the trajectory of each vehicle describes its movement history. We study joins on two sets of trajectories, generated by two sets M and R of moving objects. For each entity in M, a join returns its k nearest neighbors from R. We examine how this query can be evaluated in cloud environments. This problem is not trivial, due to the complexity of the trajectory, and the fact that both the spatial and temporal dimensions of the data have to be handled. To facilitate this operation, we propose a parallel solution framework based on MapReduce. We also develop a novel bounding technique, which enables trajectories to be pruned in parallel. Our approach can be used to parallelize existing single-machine trajectory join algorithms. We also study a variant of the join, which can further improve query efficiency. To evaluate the efficiency and the scalability of our solutions, we have performed extensive experiments on large real and synthetic datasets.

Index Terms—Nearest neighbor, Trajectory join, Big trajectory data, MapReduce.

1 Introduction

In emerging systems that manage moving objects, a tremendous amount of *trajectory data* is often produced. In a location-based service (LBS), for instance, the positions of mobile phone users or vehicles are constantly captured by GPS receptors and mobile base stations [1], [2]. The location information constitutes a trajectory, which depicts the movement of an entity in the past. In natural habitat monitoring, scientists obtain location information of wild animals by attaching sensors to them. This movement history information, or trajectory data, facilitates the understanding of the animals' behaviours [3]. Figure 1(a) gives six trajectories, each of which is constructed by connecting three recorded locations.

Due to the increasing needs of managing trajectory data, the study of *trajectory databases* has recently attracted a lot of research attention [3]. One of the fundamental queries for this database is the *join* [4], [5], [6]. Given two sets M and R of trajectory objects, a join operator returns entity sets from M and R, that exhibit proximity in space and time. To illustrate this query, let us consider Figure 1(a), where two sets of trajectory objects, namely M=$\{m_1, m_2, m_3\}$ and R=$\{r_1, r_2, r_3\}$, are shown. Each trajectory is constructed by connecting the locations collected at time instants t_1, t_2 and t_3, where $t_1 < t_2 < t_3$. For each trajectory, the small white dot represents the position recorded at t_1. The result of joining M and R is demonstrated in Figure 1(b). For each object $m_i \in M$, the two counterparts in R that are the nearest neighbors of m_i in $[t_2, t_3]$ are returned. In this paper, we adapt the k*-nearest neighbor* metric [5], [6], as the joining criterion of M and R. That is, the k objects in R that have the shortest distances from each object in M are returned, by adopting the *closest-point-of-approach* [4]. In this example, within the time interval $[t_2, t_3]$, the 2-NNs of m_1 are r_1 and r_2.



(a) Two sets of trajectories

	k nearest neighbors
m_1	r_1, r_2
m_2	r_3, r_2
m_3	r_2, r_1

(b) Join results

Fig. 1. Illustrating the k-NN join (k=2, $[t_2, t_3]$).

The trajectory join query can be used in a wide range of applications, including business analysis, military applications, celestial body relationship analysis, battlefield analysis, and computer gaming [6], [7], [8]. Consider two competitor companies that provide flights in the same geographical area. Let A and B be the sets of flight routes of these two companies. By joining A and B, we can retrieve for each route $a \in A$, the k routes in B that were the closest to a in a specific time interval. These results could be further analyzed by the company that manages A and help her to answer questions like: Is there any plane in B that flew very close to A's flights and might cause safety concerns? Is there any route of B that resembles a, and charges a lower fare? In military applications, consider the two sets C and D of trajectories for military units (e.g., soldiers, vehicles and tanks) belonging to two rival countries. By joining C and D, the k entities in D closest to those of C can be evaluated. This is useful for the C's army to study the movement patterns of D, and study whether D is performing inspection on C, or planning a military action.

The trajectory join query can also be useful in astronomy databases, where the trajectory information of objects in the outer space is abundant [7]. For example, the Hubble Space Telescope of NASA generates 140GB of data about movements of stars and asteroids on a weekly basis [9], from which precise trajectory information is extracted [10]. In the Sloan Digital Sky Survey (SDSS) project, up to 20TB of location data about millions of outer-space objects are collected every night [11]. These huge

- *Y. Fang, R. Cheng, W. Tang, and X. Yang are with the Department of Computer Science, The University of Hong Kong, Hong Kong.*
- *S. Maniu is with Université Paris-Sud, Orsay, France.*
  E-mail: {yxfang, ckcheng, wbtang, xyang2}@cs.hku.hk, maniu@lri.fr

Manuscript received December 20, 2014.

amounts of data enable the analysis of the behavior of outer-space objects, such as discovery of meteors that were close to the Earth [12], and evaluating frequency evolution [13]. These tasks, which require the analysis of proximity among trajectory objects, can be facilitated by the trajectory join. For example, given two groups A and B of asteroids the query returns the identities of asteroids from B that have been close to those in A.

Despite the usefulness of trajectory joins, evaluating them is not trivial. A simple solution is to evaluate a k-NN query for every object in M. However, since a trajectory object describes the movement of points in space and time, its data structure can be complex and expensive to handle. The problem is worsened when the sizes of the trajectory object sets to be joined are large. To evaluate joins on large trajectory datasets efficiently, researchers have previously studied fast algorithms and data structures [4], [5], [6]. However, these approaches run trajectory joins on a single machine only, whose computation, memory, and disk capabilities are limited. As discussed before, extremely large trajectory data have become increasingly common. Two trajectory datasets [1], [2], for instance, consist of over one billion location values. For evaluating joins on these large data, a single machine is no longer sufficient. In this paper, we study efficient trajectory join algorithms in parallel and distributed environments. We choose MapReduce as the platform for our study, since it provides decent scalability and fault tolerance for very large data processing [14].

Designing trajectory join algorithms on MapReduce is technically challenging. This is because MapReduce is a shared-nothing architecture. Existing single-machine solutions often rely on an index (e.g., R-tree) built on top of the whole dataset (e.g., [5], [6]). As discussed in [15], constructing and maintaining an index in MapReduce can be costly. In this paper, we develop a solution framework that exploits the shared-nothing architecture, without using an index. We first partition the given trajectories of M and R into "sub-trajectories", which are distributed to different computation units. For each partition of sub-trajectories, we develop a *time-dependent bound* (or *TDB* in short). The TDB is a time-dependent circular region containing the (candidate) objects in R, which can be the k nearest neighbors of objects in M, in the same partition. Based on the TDB, we retrieve R's candidates, and join them with M's sub-trajectories. The join results of the partitions are finally merged.

Our solution can easily adopt single-machine join algorithms in its framework. In the paper, we will study how our approach parallelizes two single-machine solutions using MapReduce. Moreover, as we will discuss, the TDB is a function of time, and it changes according to the positions of the objects involved. While computing a TDB is not straightforward, we show that it is possible to develop a theoretically efficient algorithm to evaluate the TDB in parallel. The effort of developing TDB is justified by our experiments, which show that TDB significantly reduces the number of candidates to be examined.

We further propose two methods to improve our solutions. First, we use hash functions to distribute trajectory objects to computing units more uniformly, in order to achieve better load balancing. Second, we study a variant of the k-NN join, called (h,k)-NN join, which returns h objects in M, together with their corresponding k-NN in R, aggregated using a monotonic function, e.g., min, max, sum or avg [16]. This query, which represents the h sets of "most important" k-NNs in the join of M and R, is important when the query user is interested in the k objects in R that are spatially close to those in M. For example, given the trajectories of a large number of asteroids, the (h,k)-NN join returns h asteroids in set M, and the h sets of k asteroids from R that are the closest to the h asteroids in M. The objects returned from R may have a higher chance to collide with the h objects in M. Since only h sets of k-NN objects are returned, a query user can focus on a smaller set of results (compared with the original join problem where the $|M|$ sets of k-NN objects are returned). We have also developed a tighter pruning bound for the (h,k)-NN join. In our experiments, the (h,k)-NN join is much faster than the k-NN join.

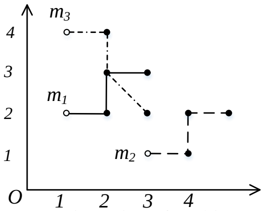
The rest of this paper is organized as follows. In Section 2, we review the related work. Section 3 discusses the preliminaries. In Section 4 we study the framework of our solution. In Sections 5 and 6, we present the detailed solution of the k-NN join. In Section 7, we study the solution of (h,k)-NN join. Section 8 discusses the experimental results. We conclude in Section 9.

2 Related Work

A substantial amount of research on nearest neighbor query for trajectory objects has been performed. In [5], the authors classify four types of k-NN queries, along two dimensions. First, the query object q can be a stationary point or a trajectory. Second, given a time interval U, the query result can be continuous or a snapshot. Our join query is extended from the query class (*moving query object, snapshot result*) in [5]: for each trajectory m in set M, return k trajectories in R that are the closest to m during U. In [17], the nearest neighbor of every point on a line segment has been studied; [18] addressed concurrent continuous spatio-temporal queries; and [8] examined the k nearest and reverse k NN queries. The main difference between our solution and [8], [17], [18] is that for a given query trajectory object, we compute the k trajectory objects whose distances to the query are minimal a particular time instance, while these works focus on returning the k NNs at every time instance.

There are also many studies on join operation for trajectories. In [4], an adaptive join algorithm is proposed for closest-point-of-approach join, which is based on the *sweep line algorithm* [19]. Given two objects, their minimum distance is defined to be achieved at their closest point. Also in [6] a broad class of trajectory join operations are studied, including distance join and k-NN join. We also note many studies which focus on trajectory similarity query based on metrics such as DTW, LCSS [20], ERP [21] and EDR [21]. The main difference between our metrics and the above is that here we focus on objects' spatial distances, while they focus on trajectories' shape similarities. We illustrate this in Figure 2, with the trajectories of 3 objects, whose positions are collected at the same 4 time instances (the white dots denote the start points). Let us use EDR in this example. This similarity metric computes the edit distance between pairs of points from the trajectories after spatial normalization. For each object, we list other objects based on their distance values on EDR and our metric in ascending order in Figure 2(b). Let us take m_1 as an example. Since compared with m_3, m_2's trajectory shape is more similar with that of m_1 and thus the EDR values between m_1 and m_2 is lower than that between m_1 and m_3. However, m_1 is spatially closer to m_3 than m_2. Hence, our metric is more useful in situations in which spatial proximity is an important factor.

These existing join algorithms, however, are designed to be executed on a single machine, and hence are inefficient on large datasets. A natural way to extend them for handling large-scale



(a) Trajectories of 3 objects

	EDR	Our
m_1	m_2, m_3	m_3, m_2
m_2	m_1, m_3	m_3, m_1
m_3	m_1, m_2	m_1, m_2

(b) Rank results

Fig. 2. Examples of EDR and our distance metric.

data is to use parallel computing. There exist a few parallel computing paradigms including MapReduce [14], Pregel, Spark and Shark [22]. Out of these, MapReduce is one of the most widely used and performs best for batch processing queries – such as joins – and we study answering k-NN join queries using the MapReduce here. As we will discuss, the naive way to extend single-machine join algorithms for MapReduce is not efficient enough for large-scale trajectories due to its high computational cost.

Recently, many different kinds of join operations have been studied using MapReduce. For example, in [23] the set-similarity join is answered efficiently using MapReduce, and in [24] the multi-way theta-join query is studied. In [25] efficient algorithms for k-NN join are presented using MapReduce, but they mainly return approximate join results for sets of points, while our k-NN join returns the exact result. [26] designs an effective mapping mechanism that exploits pruning rules for distance filtering. However, since they do not deal with the temporal dimension, it is not clear how they can be applied to the data of trajectory objects.

3 Preliminaries

In this section we formally introduce the data model, problem definitions, single-machine solutions, the MapReduce framework, and a basic parallel solution using MapReduce.

3.1 Data Model

For ease of presentation, we consider in the following objects – or *trajectories* – in a $d \times d$ two-dimensional space. Note, however, that our methods can easily be applied for multi-dimensional space. Table 1 summarizes the symbols used in this paper.

Definition 1. A trajectory tr of an object is a tuple composed of the object's id and a list of locations $(q(t_1), q(t_2), \cdots, q(t_l))$. Each point $q(t)$ is represented by a triple (x, y, t), where x and y are the positions along x and y coordinates, and t is the timestamp of this location.

We denote the timestamps of the first and last points of tr as $tr.s$ and $tr.e$. A sub-trajectory of tr is a part of tr, such that the timestamps of its first and last points are in $[tr.s,\ tr.e]$. In line with the linear trajectory model proposed [4], [6], we assume that an object moves along the line segment $\overline{q(t_i)q(t_{i+1})}$ between any two consecutive points $q(t_i)$ and $q(t_{i+1})$ with a constant speed. This allows us to compute the position of the object at any given time t in the time interval $[t_i, t_{i+1}]$, thereby evaluating the distance between two trajectories. Our methods can be extended to support other speed functions, as long as the function of finding the object's location with a given time is constant time.

TABLE 1
Summary of Notations

Notation	**Meaning**
$M(R)$	a set of trajectory objects
$m(r)$	a trajectory object from $M(R)$
tr	a trajectory
$tr.id$	the id of the object with trajectory tr
$tr.q(t_i)$	a point of tr whose time instance is t_i
$tr.s, tr.e$	the start and end time instances of tr
l	the total number of points in tr
T	the number of temporal partitions
N	the number of spatial partitions
H	the number of trajectory groups after hashing
p_i	the central point of i-th spatial (grid) partition
Tr_i^M	trajectories in i-th grid generated by objects from M
$\|Tr_i^M\|$	the number of trajectories in a partition Tr_i^M
C_i^R	a set of candidate trajectories from R for Tr_i^M
G_i^M	a group of trajectories from M whose hash values are i

Definition 2. The minimum distance between a point p and a line segment $\overline{q(t_i)q(t_{i+1})}$, is defined as:

$$MinDist\left(p, \overline{q\left(t_i\right) q\left(t_{i+1}\right)}\right) = \min\{|p,q| \mid q \in \overline{q(t_i)q(t_{i+1})}\} \tag{1}$$

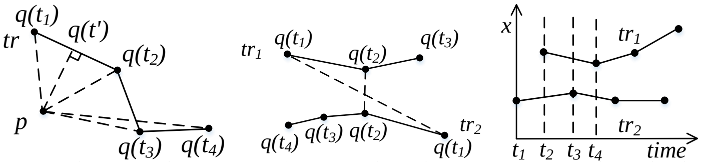
where q is a point lying on the line segment $\overline{q(t_i)q(t_{i+1})}$, and $|p,q|$ is the Euclidean distance between points p and q.

Without loss of generality, our algorithm can be easily extended for other trajectory models [3] and distance measures such as network distance, Manhattan distance, etc.

Definition 3. The minimum distance between a point p and a trajectory tr with l points is defined as:

$$\begin{aligned} MinDist(p, tr) = \min\{\ & MinDist\left(p, \overline{tr.q\left(t_i\right) tr.q\left(t_{i+1}\right)}\right) \\ & \mid i \in \{1, \cdots, l-1\}\} \end{aligned} \tag{2}$$

Similarly, we can define the maximum distance $MaxDist$ $(p, \overline{q(t_i)q(t_{i+1})})$ between a point p and a line segment $q(t_i)q(t_{i+1})$, and the maximum distance $MaxDist(p, tr)$ between a point p and a trajectory tr.



(a) A point & a trajectory (b) Two trajectories (c) Computing MinDist

Fig. 3. Examples of minimum and maximum distances

Example 1. In Figure 3(a), we can easily observe that $MinDist(p,\overline{q(t_1)q(t_2)})$=$|p, q(t')|$, $MaxDist(p, \overline{q(t_1)q(t_2)})$=$|p, q(t_2)|$, $MinDist(p,tr)$=$|p, q(t')|$ and $MaxDist$ $(p,\ tr)$=$|p, q(t_4)|$.

Definition 4. The minimum distance between two (objects with) trajectories tr_i and tr_j is defined as:

$$MinDist(tr_i, tr_j) = \min\{|tr_i.q(t), tr_j.q(t)| \mid t \in \Delta t\}, \tag{3}$$

where $\Delta t = [tr_i.s, tr_i.e] \cap [tr_j.s, tr_j.e]$.

To understand how to compute $MinDist(tr_i, tr_j)$, let us use Figure 3(c), which shows two trajectories, tr_1 and tr_2. Consider

$[t_2, t_4]$, where tr_1 and tr_2 overlap in time. We first obtain the "sub-intervals" within $[t_2, t_4]$ such that the line segments from tr_1 and tr_2 overlap. These sub-intervals are $[t_2, t_3]$ and $[t_3, t_4]$. For each sub-interval, we compute the minimum distance between the two respective line segments. This can be done by using the method in [4], which expresses this distance as a quadratic function of time and finds its minimum value. Then, $MinDist(tr_1, tr_2)$ is the lowest value of the two minimum distances obtained at $[t_2, t_3]$ and $[t_3, t_4]$. Since the number of sub-intervals between two trajectories is $O(l)$, and the cost of computing the minimum distance for each sub-interval is $O(1)$, the complexity of evaluating $MinDist(tr_i, tr_j)$ is $O(l)$.

We can define $MaxDist(tr_1, tr_2)$ in a similar way, as the maximum distance between two trajectories tr_1, tr_2.

Example 2. In Figure 3(b), we can easily observe that $MinDist(tr_1, tr_2)$ = $|tr_1.q(t_2), tr_2.q(t_2)|$ and $MaxDist$ (tr_1, tr_2) = $|tr_1.q(t_1), tr_2.q(t_1)|$.

Definition 5. Given a trajectory object m and a set of trajectory objects R, the k $nearest\ neighbors$ of m are the k objects from R, whose minimum distances with m are the smallest.

Given a trajectory object m and the k minimum distances $d_1^m, \ldots, d_k^m$ to its k nearest neighbors, we consider a monotonic aggregate function f (e.g., max, min, sum or average of the k distance values [16]) on m. Without loss of generality, we use $f(m)$ as the maximum of these k distance values, i.e.,

$$f(m) = \max\left\{d_i^m \mid i \in \{1, \cdots, k\}\right\}. \tag{4}$$

3.2 Problem Definition

Given two sets M and R of trajectory objects in the time domain $[T_s, T_e]$, we study two problems:

Problem 1. k**-NN join:** Given an integer k and a query time interval $[t_s, t_e] \subseteq [T_s, T_e]$, return the k nearest neighbors from R for each object in M during $[t_s, t_e]$.

Problem 2. (h, k)**-NN join:** Given two integers h, k, a query time interval $[t_s, t_e] \subseteq [T_s, T_e]$, and a monotonic aggregation function f, return the h objects of M with the smallest f values on their k nearest neighbors from R during $[t_s, t_e]$. For each of these h objects, return its k nearest neighbors from R.

Example 3. In Figure 1(a), M=$\{m_1, m_2, m_3\}$, R=$\{r_1, r_2, r_3\}$, and the time domain is $[t_1, t_3]$. Let k=2, $[t_s, t_e]$=$[t_2, t_3]$, and h=1. The results for k-NN join (Problem 1) are shown in Figure 1(b). For example, the 2 closest trajectories of m_1 are r_1 and r_2, whose their minimum distances are obtained at t_3. For (h, k)-NN join (Problem 2), $f(m_1) = \max\{d_1, d_2\} = d_2$, while $f(m_2)$ and $f(m_3)$ are larger than d_2. Hence, the result of (h, k)-NN join is $\{m_1, \{r_1, r_2\}\}$. Intuitively, $\{r_1, r_2\}$ is the set of the "most important" k-NNs. Problem 2 allows users to focus on a smaller list of results (i.e., one k-NN set for m_1), instead of reading the list of k-NNs for all the three objects in M. It also enables a faster solution, as we will discuss later.

3.3 Single-machine Solutions

We now discuss two single-machine solutions for solving k-NN joins, namely *brute force* (**BF**) and *sweep line* (**SL**).

Brute Force (BF): This method uses nested loops to perform the join. It first selects all the sub-trajectories that appear in $[t_s$, $t_e]$. Then it computes the minimum distance between each pair of trajectory objects (i.e., one from M and the other one from R), and selects the k nearest neighbors for each object of M. This method is costly, as it may need to process the Cartesian product of M and R.

Sweep line (SL): This method, proposed in [4], [19], first sorts all the points that belong to objects from R in ascending order of their timestamps. For each trajectory $tr \in M$, SL conceptually "sweeps" a line (by scanning the points in R in ascending order of their timestamps), and computes the minimum distance between tr and each object in R that overlap in time. The k nearest neighbors are found by considering all these distance values. In Figure 3(c), we suppose that $tr_1 \in M$ and $tr_2 \in R$. Finding the k nearest neighbors of tr_1 necessitates the evaluation of $MinDist(tr_1, tr_2)$. To compute $MinDist(tr_1, tr_2)$, a (dotted) line is conceptually moved, starting from t_1. As this line moves, the minimum distances between sub-trajectories of tr_1 and tr_2 that overlap in the same time interval (e.g., $[t_2, t_3]$) are computed. The value of $MinDist(tr_1, tr_2)$, which is the least of the minimum distances between the sub-trajectories of tr_1 and tr_2, can be used to decide whether tr_2 is the k-NN of tr_1.

The time complexities of BF and SL are both $O(|M||N|l)$. However, if a pair of trajectories does not have any temporal intersection, SL will not consider them; therefore, it is more efficient in the average case.

3.4 MapReduce Framework

MapReduce [14] is a popular paradigm for processing large data on share-nothing distributed clusters. It consists of two functions `map` and `reduce`. The `map` function takes a key-value pair and outputs a list of key-value pairs, i.e., `map`$(k_1, v_1)\rightarrow$ *list*(k_2, v_2). The `reduce` function takes a list of records with the same key as input and outputs a list of key-value pairs, i.e., `reduce`$(k_2$, *list*$(v_2)) \rightarrow$ *list*(k_3, v_3). In a MapReduce job, when all the `map` functions have finished, all the intermediate results are grouped and shuffled to the `reduce` functions. By default, each map task processes a split of data with the size equal to the block size of its distributed file system, HDFS. In a MapReduce job, the number of map tasks equals to the number of splits, while number of reduce tasks can be set by the users.

3.5 A Basic Parallel Solution Using MapReduce (BL)

We now introduce a baseline parallel solution of k-NN join using MapReduce, i.e., **BL**, which has two MapReduce jobs. In the first job, it divides objects in M and R into a list of disjoint subsets randomly in the `map()`, and then joins each pair of subsets using a single-machine solution – e.g., BF or SL – in the `reduce()`. Since the k nearest neighbors of an object may be in several subsets of R, a second job where the k nearest neighbors are selected, from the results of the first MapReduce job. The main drawback of BL is its high computational cost, since each pair of trajectories from M and R needs to be enumerated.

4 Solution Framework

To overcome the drawback of BL, we propose a new parallel solution framework, which allows pruning of trajectories during the querying process. Our framework consists of two phases, namely **preprocessing phase** and **query phase**.

Figure 4 illustrates its workflow. The preprocessing phase needs to be conducted for only once, while the query phase is

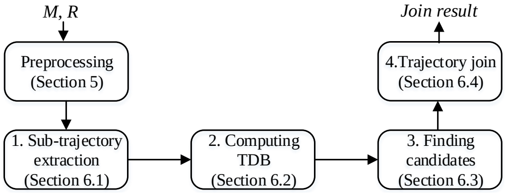

M, R
Preprocessing (Section 5)
1. Sub-trajectory extraction (Section 6.1)
2. Computing TDB (Section 6.2)
3. Finding candidates (Section 6.3)
4.Trajectory join (Section 6.4)
Join result

Fig. 4. The workflow of our framework

invoked once a k-NN join query arrives. In the preprocessing phase, we partition trajectories in temporal and spatial dimensions. In the query phase, we propose a four stage approach to answer a join query:

1) **Sub-trajectory extraction.** In this stage, we extract all the sub-trajectories appearing in $[t_s, t_e]$. Then we collect relevant statistics from each spatial partition. We also select trajectories, which serve as *anchor trajectories*, if this partition of trajectories is generated by objects from R. We denote the set of sub-trajectories generated by objects from $M(R)$, in the $i(j)$-th grid, as $Tr_i^M(Tr_j^R)$.
2) **Computing TDB.** In this stage, we compute the *time-dependent upper bound* (TDB) of Tr_i^M using the collected statistics and the anchor trajectories.
3) **Finding candidates.** For each partition Tr_i^M, we use its TDB to find a set, C_i^R, of *candidate trajectories* generated by objects from R. The candidate trajectories are sets of trajectories – ideally, minimal in size – which must contain *all* the k nearest neighbors of objects in M which cross the i-th spatial grid.
4) **Trajectory join.** For each partition Tr_i^M, we join it with C_i^R using a single-machine algorithm, (e.g., BF or SL).

We denote the above approach as **GN** (Grid with No load balance). Even though GN achieves greater efficiency by using TDB, it may not able to achieve good load balance, due to the skewness of the distribution of the trajectory objects. By using uniform partitioning of the space, we may encounter grids which contain many objects while others contain very few objects. To alleviate this issue, we improve the load balance of GN under the same framework, by using a load balance strategy, which redistributes all the objects using some hash functions. We denote this new approach as **GL** (Grid with Load balance). It is worth noting that our framework can support various spatial partition methods such as quadtree and Voronoi diagram [26].

The preprocessing phase and each stage of the query phase are computed using a MapReduce job, in a sequential workflow. The purpose of the `map()` and `reduce()` for each stage is summarized in Table 2. We discuss the preprocessing in Section 5. We detail the stages of GN and GL in Section 6.

5 The Preprocessing Phase

The preprocessing phase is mainly used to partition the data, both temporally and spatially. Since the query time interval is usually smaller than the time interval of the entire data, we propose to partition the trajectories using equal-length time intervals for efficient sub-trajectory extraction. Spatially, the trajectories are partitioned using equal-sized grids. For each trajectory, we first conduct temporal partitioning and obtain a list of sub-trajectories, and then we conduct spatial partitioning for each sub-trajectory.

TABLE 2
Details of each phase and stage

Phase/Stage	Function	Main work
Preprocessing	Map	conduct temporal partitioning
	Reduce	conduct spatial partitioning
Stage 1	Map	extract sub-trajectories
	Reduce	collect statistics and anchor trajectories
Stage 2	Map	compute $MaxDist(p_i, achTr)$
	Reduce	compute the TDB, i.e, $u_i(t)$, of Tr_i^M
Stage 3	Map	find candidates of Tr_i^M
	Reduce	collect the candidates of Tr_i^M
Stage 4	Map	trajectory join
	Reduce	output k nearest neighbors

1. Temporal partitioning. Suppose that the number of intervals is T. We define a list of time intervals: $[T_s, T_s + \Delta t]$, $[T_s + \Delta t, T_s + 2\Delta t], \cdots, [T_e–(T–1) \cdot \Delta t, T_e]$, where Δt=$(T_e– T_s)/T$. For each trajectory tr, we compute its intersections with these intervals; if an intersection point between tr and a time interval is not one of the points of tr, it is added to tr. We then split tr into a list of sub-trajectories according to these time intervals.

2. Spatial partitioning. We first partition the space using N equal-size grids (We will examine the setting of T and N in our experiments in Section 8). Given the i-th grid, we denote its *central point* p_i, which is the centre of the i-th grid, with the same distance to the four corners of the grid. To partition a trajectory tr, we compute its intersection points with the grids, and insert them into tr, if they do not belong to the points of tr. Finally, tr is partitioned into a group of sub-trajectories, each of which is in its corresponding grid. For each sub-trajectory $subTr$, we assign it a key M_i, if it is from an object from M; otherwise we give it a key R_i, where i is the i-th grid containing $subTr$. Notice that some sub-trajectories of a trajectory may lie in the same grid.

Note that after the trajectories of M and R are loaded into HDFS, we only need to conduct preprocessing for once. We pre-process the trajectories using a MapReduce job, where the temporal and spatial partitioning are performed in the `map()` and `reduce()` respectively, as shown in Algorithm 1.

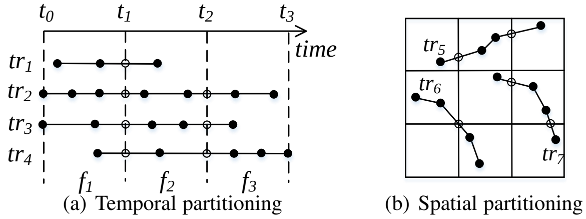
Algorithm 1 Preprocessing

```
1:  procedure MAP($k_1$, $v_1$)
2:      $trList \leftarrow$ TEMPORALPARTITION($v_1$);
3:      for each $tr \in trList$ do
4:          $k_2 \leftarrow tr.InterIndex$, $v_2 \leftarrow tr$;
5:          OUTPUT($k_2$, $v_2$);
6:  procedure REDUCE($k_2$, $v_2$)
7:      for each $tr \in v_2$ do
8:          $subTrList \leftarrow$ SPATIALPARTITION($tr$);
9:          for each $subTr \in subTrList$ do
10:             $k_3 \leftarrow k_2$, $v_3 \leftarrow (subTr.spatioKey, subTr)$;
11:             OUTPUT($k_3$, $v_3$);
```

Map. The input of `map()` is a pair (k_1, v_1), where v_1 is a trajectory. Then this trajectory is split into a list of sub-trajectories according to the predefined time intervals (line 2). The output is list of (k_2, v_2) pairs (lines 3-5), where the k_2 is the identifier of the time interval and the v_2 is a sub-trajectory.

Reduce. After shuffling, trajectories with a same key are sent to a same `reduce()`. For each trajectory, we do spatial partitioning and obtain a list of sub-trajectories (lines 7-8), each of which is assigned a spatial key M_i (R_i), if it is in the i-th grid and the corresponding object is from M (R)(lines 9-10).

In the output of this MapReduce job, we generate trajectories and output them into the same file if they occur in the same time interval. Hence, we obtain T files $f_1, f_2, \cdots, f_T$, where the time intervals of trajectories in f_i are in $[T_s + i \cdot \Delta t, T_s + (i+1) \cdot \Delta t]$. The time and space complexities of `map()` are $O(l+T)$, since a trajectory is split into at most T sub-trajectories. Both the time and space complexities of `reduce()` are $O((|M|+|R|)(l+N))$.



(a) Temporal partitioning

(b) Spatial partitioning

Fig. 5. Examples of preprocessing

Example 4. In Figure 5, the black and white dots represent the original and inserted points respectively. For instance, tr_1 is split into two sub-trajectories, one appearing in $[t_0, t_1]$ and the other appearing in $[t_1, t_2]$, and tr_6 is partitioned using the spatial grids and two sub-trajectories are obtained, each of which appears in a single grid.

6 The Query Phase

In this section, we first introduce the four stages of the query phase of GN respectively, and then discuss GL.

6.1 Sub-trajectory Extraction

When a join query is asked, we first need to locate the files relevant for the join operation and then launch a MapReduce job with these files as an input. In the mapper we retrieve all the sub-trajectories which intersect with $[t_s, t_e]$. In the reducer, we collect some statistics and anchor trajectories, which are used for computing the TDB for each partition.

Algorithm 2 gives the details. We first locate files f_c,f_{c+1}, $\cdots$, $f_{c'}$ (line 2), which have time intervals overlapping with $[t_s, t_e]$, where c =$\lfloor t_s/\Delta t \rfloor$ and c'=$\lceil t_e/\Delta t \rceil$.

Algorithm 2 Stage 1: Sub-trajectory extraction

```
1: procedure MAP-SETUP($t_s$, $t_e$)
2:     locate files $f_c$, $f_{c+1}$, $\cdots$, $f_{c'}$;
3: procedure MAP($k_1$, $v_1$)
4:     $subTr$ ← $v_1$.subTraj($t_s$, $t_e$);
5:     $k_2$ ← $k_1$; $v_2$ ← $subTr$; OUTPUT($k_2$, $v_2$);
6: procedure REDUCE($k_2$, $v_2$)
7:     parse the set label $L$ from $k_2$; $Tr_i^L$ ← $v_2$;
8:     compute $sT(Tr_i^L)$, $eT(Tr_i^L)$, $maxU(Tr_i^L)$;
9:     if $L$ = "$R$" then
10:        $achList$ ← SELECTANCHOR($Tr_i^L$, $sT(Tr_j^R)$, $k$);
11:    output $maxU(Tr_i^L)$, $sT(Tr_i^L)$, $eT(Tr_i^L)$, $achList$, $Tr_i^L$;
```

Map. The input of `map()` is a pair (k_1, v_1), where v_1 is a trajectory, and k_1 is its key (i.e., M_i or R_i). We extract v_1's sub-trajectory, $subTr$, which appears in $[t_s, t_e]$ and is involved in the join (line 4), and output it (line 5).

Reduce. We first parse the set label L, which can be either M or R from k_2 (line 7). Then we collect some statistic information and anchor trajectories (lines 8-11).

In terms of statistics collected, we first collect the minimum start time and maximum end time of all the trajectories in Tr_i^L (line 8), as shown in Equation (5). Moreover, for all the trajectories, we compute their maximum distances to the central point p_i, and collect their maximum value, as shown in Equation (6).

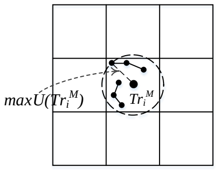


Fig. 6. Statistic collection

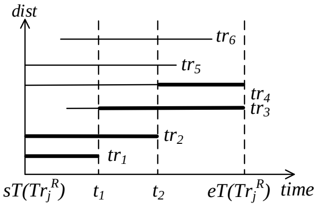


Fig. 7. Anchor trajectory selection

$$sT(Tr_i^L) = \min_{tr \in Tr_i^L} tr.s, \quad eT(Tr_i^L) = \max_{tr \in Tr_i^L} tr.e \qquad (5)$$

$$maxU(Tr_i^L) = \max_{tr \in Tr_i^L} MaxDist(p_i, tr) \qquad (6)$$

To facilitate our computations, we now introduce a new data structure, namely spatiotemporal-unit (abbreviated as *st-unit*). A **st-unit** is a triple ($dist$, $startT$, $endT$), where $dist$ is a distance value, $startT$ and $endT$ are the start and end time of a time interval $[startT, endT]$. Let p_i be the centre of the i-th grid. For each partition Tr_i^L, we can form a st-unit $(maxU(Tr_i^L), sT(Tr_i^L), eT(Tr_i^L))$, where $maxU(Tr_i^L)$ bounds the maximum distances from p_i to all the trajectories during this time interval.

In addition, if L is R, we need to collect *anchor trajectories* from Tr_j^R (lines 9-10). An anchor trajectory is a trajectory in Tr_j^R in the time interval $[sT(Tr_j^R), eT(Tr_j^R)]$. In line 10 (`selectAnchor`), we select up to k anchor trajectories from Tr_j^R, for the purpose of computing the TDB (Section 6.2). Conceptually, for any time instance $t \in [sT(Tr_j^R), eT(Tr_j^R)]$, `selectAnchor` chooses k trajectories that are spatially close to p_i if there are at least k objects in Tr_j^R; otherwise all of them will be selected. The next example illustrates this process. For details, please refer to Appendix A.

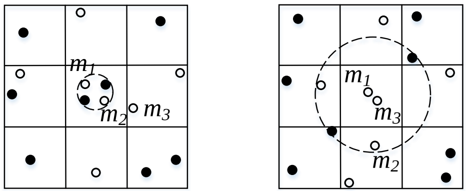
Example 5. Figure 6 gives an example of computing $maxU(Tr_i^M)$ of Tr_i^M, which contains 2 trajectories. Figure 7 gives an example of 6 trajectories in Tr_j^R (k=2). Since each trajectory tr can be formed as a st-unit ($MaxDist(p_j, tr)$, $tr.s$, $tr.e$), we plot a line segment for each trajectory, where its length is $|tr.e–tr.s|$ and distance to $time$-axis is $MaxDist(p_j, tr)$. We can observe that at any time instance, there are at least 2 trajectory objects in the j-th grid. We collect the trajectories which correspond to the wide lines in the figure as anchor trajectories.

The time and space complexities of `map()` are $O(\log l)$ and $O(l)$ respectively, since we can use binary search to find the sub-trajectory. The computation of $sT(Tr_i^L)$, $eT(Tr_i^L)$ and $maxU(Tr_i^L)$ can be performed linearly by scanning all the trajectories. The time complexity of finding anchor trajectories is $O(|Tr_i^L|^2 l)$, since we need to find one from Tr_j^R each time. Thus, the overall time and space complexities of `reduce()` are $O(|Tr_i^L|^2 l)$ and $O(|Tr_i^L| l)$ respectively.

6.2 Computing TDB

Let us now introduce the *time-dependent bound* (or *TDB*), which is used to prune trajectories during the join operation. Given a set

S of objects whose sub-trajectories are in Tr_i^M, a TDB is a time-dependent circle $c(t)$, with $t \in [sT(Tr_i^M), eT(Tr_i^M)]$, such that the k nearest neighbors of S at time t is contained in $c(t)$. In Figure 8, the white and black dots denote the objects from M and R respectively. Let us consider the TDB for the objects whose trajectories overlap the grid in the centre. Figure 8(a) shows that at time t, $c(t)$ contains the k nearest neighbors of m_1 and m_2 (where k=2). The area of $c(t)$ is small. However, at another time instance t', the area of $c(t')$, which bounds the 2 nearest neighbors of m_1 and m_3, is much larger than $c(t)$ (Figure 8(b)).



(a) time t

(b) time t'

Fig. 8. Illustrating the change of TDB with time.

As shown in the example above, due to the movement of objects, the TDB for Tr_i^M changes with time. Hence, it is not straightforward to compute the TDB. Our main idea is to first evaluate the maximum distance $v_i(t)$ of the objects of R that are the k nearest neighbors of p_i at time t. We then compute the TDB of Tr_i^M by using $v_i(t)$ and the area of the grid that bounds Tr_i^M, as summarized below:

1) We first compute TDB of p_i, i.e., $v_i(t)$, by using the maximum distances from p_i to all the anchor trajectories.
2) We compute the TDB of Tr_i^M, i.e., $u_i(t)$, by using the TDB of p_i and $maxU(Tr_i^M)$.

To compute $v_i(t)$, we can use spatial indexes for fast access, e.g., R-trees. As discussed before, however, is is difficult to maintain such indexes using MapReduce [15], which is a share-nothing framework. Another way is to enumerate all the distances between p_i and all the objects in R, and then compute a bound. However, the computational cost is very high, since $|R|$ can be very large.

Step 1. Compute $v_i(t)$. To compute $v_i(t)$, we first form a st-unit ($MaxDist(p_i,\ achTr)$,$sT(Tr_i^M)$,$eT(Tr_i^M)$) for each anchor trajectory $achTr$. Given such a list of st-units, we can compute $v_i(t)$ as follows. For any time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$, we first locate the st-units, whose time intervals contain t, and then rank these st-units based on their $dist$ values in ascending order. Then we select k st-units, whose $dist$ values are minimal. Since these k st-units correspond to k objects from R, the maximum value of these k $dist$ values is thus an upper bound of p_i at t.

Since the TDB of p_i is represented by a list of st-units, it can also be rewritten as a piecewise function as follows:

$$v_i(t) = \begin{cases} d_{i,1}, & t \in [sT(Tr_i^M), t_{i,2}) \\ d_{i,2}, & t \in [t_{i,2}, t_{i,3}) \\ \cdots, & t \in [\cdots, \cdots) \\ d_{i,B}, & t \in [t_{i,B}, eT(Tr_i^M)] \end{cases} \tag{7}$$

where B is the number of st-units, $d_{i,b}$ is the upper bound distance, a constant value, when $t \in [t_{i,b}, t_{i,(b+1)})(1 \leq b \leq B)$. We call the time instances $sT(Tr_i^M)$, $t_{i,2}$, $\cdots$, $eT(Tr_i^M)$ *breakpoints*.

Figure 9 gives an example (k=2) of computing $v_i(t)$. We can observe that for any time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$, there are at least 3 st-units whose time intervals contain t. Then, our goal is to output some st-units as the TDB, which correspond to the border lines as shown in the figure. It is easy to observe that, given a time instance t, if there are many anchor trajectories close to p_i, then its upper bound distance tends to be small.

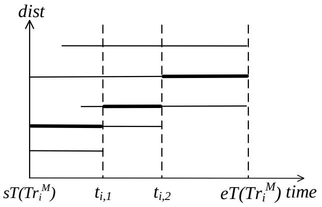


Fig. 9. Computing $v_i(t)$

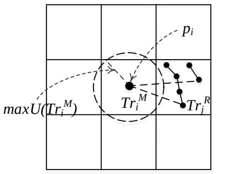


Fig. 10. Computing $u_i(t)$

To compute $v_i(t)$ more efficiently, we propose an efficient algorithm for computing $v_i(t)$ by dynamically maintaining a binary search tree (BST) [27]. We first introduce a new data structure, namely, spatiotemporal-event (abbreviated as st-$event$). A **st-event** is a triple ($time$, $dist$, $operator$), where $time$ is a time instance, $dist$ is a distance value and $operator$ is an operation, e.g., *add* or *remove*. For each st-unit u, we can create two st-events: e_1=($u.startT$, $u.dist$, *add*) and e_2=($u.endT$, $u.dist$, *remove*). The balanced binary tree [27] we used is TreeMap, which is an implementation of the balanced binary tree. In TreeMap, the key is a $dist$ value and the value is a counter, which counts the times of keys. We dynamically maintain a TreeMap of st-events according to their $operators$, and find the st-units that we need.

We give the details in Algorithm 3. We first create a list of st-events using the st-units, then sort the list by $time$ in ascending order (lines 13-19). We sweep from the earliest time instance (lines 4-5). If the next st-event has a strictly larger $time$ value, we query the k-th $dist$ from $treeMap$, form a st-unit and add it to the bound list (lines 6-9). Then, we continue updating the $treeMap$ using the st-events according to their $operators$ (lines 10-11). Finally, we obtain a list of sorted st-units, i.e., $v_i(t)$.

Algorithm 3 Computing the TDB of p_i

```
 1: procedure CompTDB($stunitList$, $k$)
 2:     $treeMap$ ← initTreeMAP<dist, counter>;
 3:     $eventList$ ← createStEvent($stunitList$);
 4:     $startT$ ← $eventList[0].time$; $TDB$ ← initList;
 5:     for each $event \in eventList$ do
 6:         if $event.time > startT$ then
 7:             $kthDist$ ← $treeMap$.findKth($k$);
 8:             $TDB$.addStUnit($startT$, $event.time$, $kthDist$);
 9:             $startT$ ← $event.time$;
10:         if $event.operator$="add" then $treeMap[event.dist]$ += 1;
11:         else $treeMap[event.dist]$ −= 1;
12:     return $TDB$;
13: procedure createStEvent($stunitList$)
14:     $eventList$ ← $null$;
15:     for each $unit \in stunitList$ do
16:         $eventList$.addEvent($unit.startT$, $unit.dist$, "add");
17:         $eventList$.addEvent($unit.endT$, $unit.dist$, "remove");
18:     sortByTime($eventList$);
19:     return $eventList$;
```

Step 2. Compute TDB. We now can compute the TDB of Tr_i^M. Specifically, we compute $u_i(t)$, the distance from each object in Tr_i^M at time t that bounds the k nearest neighbors of Tr_i^M, based on the following result:

Lemma 1. Given a partition of trajectories Tr_i^M and the TDB of the central point p_i, $v_i(t)$, the TDB for all objects having sub-trajectories in Tr_i^M, to their k nearest neighbors from R at time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$ is:

$$u_i(t) = maxU(Tr_i^M) + v_i(t). \tag{8}$$

Proof: Figure 10 illustrates the geometric intuition of the lemma. Consider an arbitrary time instance $t \in [sT(Tr_i^M), eT(Tr_i^M)]$. Suppose the k nearest neighbors of p_i are $r_j(1 \leqslant j \leqslant k) \in R$. Then $|p_i, r_j| \leqslant v_i(t)$.

Now let us consider an arbitrary object m at t, whose sub-trajectory $subTr$ is in Tr_i^M. By using triangle inequality, the distance from m to r_j is

$$|m, r_j| \leqslant |m, p_i| + |p_i, r_j|. \tag{9}$$

Since $|m, p_i| \leqslant MaxDist(p_i, subTr) \leqslant maxU(Tr_i^M)$, we have

$$|m, r_j| \leqslant maxU(Tr_i^M) + v_i(t). \tag{10}$$

The above equation implies that the distances from m to these k objects at t are bounded by $maxU(Tr_i^M) + v_i(t)$. Hence, for all objects having sub-trajectories in Tr_i^M at t, the upper bound distance to its k nearest neighbors from R is $u_i(t)$=$maxU(Tr_i^M) + v_i(t)$. □

Since $v_i(t)$ is a piece-wise function, $u_i(t)$ is also a piece-wise function, whose value changes with time. We denote the maximum and minimum values of $u_i(t)$ as $max(u_i(t))$ and $min(u_i(t))$.

In the corresponding MapReduce job, the `map()` computes the maximum distance from each anchor trajectory to each central point p_i, and the `reduce()` computes the TDB of Tr_i^M based on the maximum distances. Algorithm 4 gives the details.

Algorithm 4 Stage 2: Computing TDB

```
1: procedure MAP($k_1$, $v_1$)
2:     $achTr \leftarrow v_1$, $start \leftarrow 0$, $end \leftarrow \infty$;
3:     for $i \leftarrow 1$ to $N$ do
4:         $[start, end] \leftarrow [achTr.s, achTr.e] \cap [sT(Tr_i^M), eT(Tr_i^M)]$;
5:         $subTr \leftarrow achTr$.subTraj($start$, $end$);
6:         $maxDist \leftarrow$ MAXDIST($subTr$, $p_i$);
7:         $k2 \leftarrow M\_p_i.x\_p_i.y$, $v2 \leftarrow (maxDist, start, end)$;
8:         OUTPUT($k2$, $v2$);
9: procedure REDUCE($k_2$, $v_2$)
10:    $p_i \leftarrow k_2$, $unitList \leftarrow v_2$;
11:    $boundList \leftarrow$ COMPTDB($unitList$, $k$);
12:    for each $unit \in boundList$ do
13:        $unit.dist \leftarrow unit.dist + maxU(Tr_i^M)$;
14:    $boundList \leftarrow$ COMBINE($boundList$, $\alpha$);
15:    $k_3 \leftarrow k_2$, $v_3 \leftarrow boundList$;
16:    OUTPUT($k_3$, $v_3$);
```

Map. The input of `map()` is an anchor trajectory $achTr$. For each partition Tr_i^M, we first find $achTr$'s sub-trajectory in $[sT(Tr_i^M), eT(Tr_i^M)]$ (lines 4-5). Then we compute its maximum distance to p_i (line 6), and output a pair, where the key is "M_i" and the value is a st-unit (lines 7-8).

Reduce. After shuffling, st-units with the same key are sent to the same `reduce()`. We first compute the TDB of p_i using these st-units (line 11), resulting in a list of st-units sorted chronologically. Then we compute the TDB of Tr_i^M (lines 12-13). To reduce the number of st-units in the TDB, we merge consecutive st-units if the lengths of their time intervals are too small (line 14). Specifically, we check each st-unit u, and if $|u.endT$-$u.startT| \leq \alpha$, where α is a small predefined parameter, then we merge it with its next st-unit u'. We update u=($max(u.dist, u'.dist)$,$u.startT$,$u'.endT$) and delete u' from the list. This process is iterated until the length of time interval of each st-unit is larger than α.

The time and space complexities of `map()` are $O(Nl)$ and $O(l)$ respectively, since we need to enumerate all the central points and anchor trajectories, which can consist of the entirety of R in worst case. The operations on the balanced binary tree including insert, delete and query can be completed in $O(\log |R|)$ and combing st-units can be completed linearly without extra space cost. Thus, the time and space complexities of `reduce()` are $O(|R|log|R|)$ and $O(|R|)$ respectively.

6.3 Finding Candidates

We now study how to find a set of candidate trajectories for join, C_i^R, for each partition Tr_i^M, i.e., to list all trajectories which may be in the k-NN of trajectories of M crossing Tr_i^M. Given two partitions Tr_j^R and Tr_i^M, we check whether the trajectories of Tr_j^R are the candidates of Tr_i^R in two sequential steps: *partition check* and *trajectory check*.

Step 1. Partition check. Given two trajectory partitions, Tr_i^M and Tr_j^R, we need to check whether the whole set of trajectories in Tr_j^R is the candidate of Tr_i^M, by checking three cases:

1) none of the them are join candidates of Tr_i^M,
2) all of them are join candidates of Tr_i^M, and
3) some, but not all of them, are join candidates of Tr_i^M.

Lemma 2. Given two partitions Tr_i^M and Tr_j^R, if

$$maxU(Tr_i^M) + max(u_i(t)) \leqslant |p_i, p_j| - maxU(Tr_j^R), \tag{11}$$

then all the trajectories in Tr_j^R belong to case 1, else if

$$maxU(Tr_i^M) + min(u_i(t)) \geqslant |p_i, p_j| + maxU(Tr_j^R), \tag{12}$$

then all trajectories in Tr_j^R belong to case 2. Otherwise, the trajectories belong to case 3.

Proof: We first illustrate the geometry meaning in Figure 11. The cases between Tr_j^R with Tr_x^M, Tr_y^M and Tr_z^M are case 1, 2 and 3 respectively. We prove it as follows. Consider two arbitrary trajectories, one from Tr_i^M and one from Tr_j^R. Then consider two points p_i^M and p_j^R on the two trajectories, occurring at the same time instance. Using triangle inequality, we have:

$$\begin{aligned} |p_i^M, p_j^R| &\geqslant |p_i, p_j^R| - |p_i, p_i^M| \\ &\geqslant |p_i, p_j| - |p_j, p_j^R| - |p_i, p_i^M| \\ &\geqslant |p_i, p_j| - maxU(Tr_j^R) - maxU(Tr_i^M) \end{aligned} \tag{13}$$

$$\begin{aligned} |p_i^M, p_j^R| &\leqslant |p_i, p_j^R| + |p_i, p_i^M| \\ &\leqslant |p_i, p_j| + |p_j, p_j^R| + |p_i, p_i^M| \\ &\leqslant |p_i, p_j| + maxU(Tr_j^R) + maxU(Tr_i^M) \end{aligned} \tag{14}$$

If their minimum distance is larger than $max(u_i(t))$, then none of the trajectories in Tr_j^R are candidates. If their maximum distance is larger than $min(u_i(t))$, then all of the trajectories in Tr_j^R are candidates. For other cases, some, but not all of them are the candidates. Hence, Lemma 2 holds. □

We use the above lemma to check cases 1 and 2 first. If none of them holds, we need to perform individual trajectory check.

Step 2. Trajectory check. To explain the trajectory check, we first introduce two supporting lemmas.

Lemma 3. Given a partition Tr_i^M and a trajectory object $r \in R$ whose trajectory is tr, the lower bound distance from r to objects which cross the partition Tr_i^M is:

$$w_i(tr) = \max\{0, MinDist(p_i, tr) - maxU(Tr_i^M)\}. \tag{15}$$

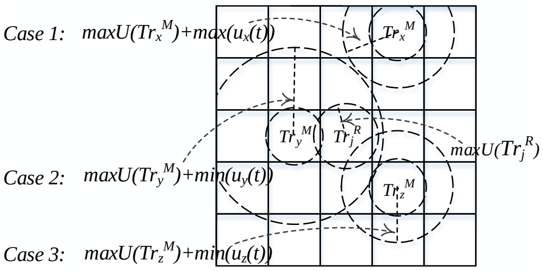

Case 1: maxU(Tr_x^M)+max(u_x(t))
Tr_x^M
Tr_y^M
Tr_j^R
maxU(Tr_j^R)
Case 2: maxU(Tr_y^M)+min(u_y(t))
Tr_z^M
Case 3: maxU(Tr_z^M)+min(u_z(t))

Fig. 11. Join candidate cases

Proof: Consider an arbitrary object m whose sub-trajectory is in Tr_i^M, and the time instance $t_{min} \in [tr.s, tr.e)$ when the minimum distance between m and r occurs. Denote m and r's positions at t_{min} as p_i^m and p_j^r respectively. Using the triangle inequality, we have:

$$\begin{aligned} |p_i^m, p_j^r| &\geqslant |p_i, p_j^r| - |p_i, p_i^m| \\ &\geqslant MinDist(p_i, tr) - maxU(Tr_i^M) \end{aligned} \tag{16}$$

For other time instance $t \in [tr_s, tr_e]$, their distance must be larger than $|p_i^m, p_j^r|$. Also, we know that $|p_i^m, p_j^r| \geqslant 0$. Hence, the lower bound distance from r to objects whose trajectories are in Tr_i^M is $\max\{0, MinDist(p_i, tr) - maxU(Tr_i^M)\}$. □

The below lemma follows directly:

Lemma 4. Given a partition Tr_i^M and its TDB $u_i(t)$, for any trajectory object $r \in R$, whose sub-trajectory tr appears in $[t_{i,b}, t_{i,(b+1)})$, where $t_{i,b}$ and $t_{i,(b+1)}$ are two consecutive breakpoints of $u_i(t)$, if the $w_i(tr) < u_i(t)$, then tr is among the candidates of Tr_i^M.

In the trajectory check step, we check each trajectory in Tr_j^R, and perform the following steps. We first collect all the breakpoints $t_{i,1}, t_{i,2}, \cdots, t_{i,B}$ of $u_i(t)$. Then, for each trajectory of Tr_j^R, we split it into a list of sub-trajectories according to the breakpoints, each of which appears in one time interval $[t_{i,b}, t_{i,(b+1)}](1 \leq b \leq B)$. For each sub-trajectory tr, we compute its lower bound distance to objects of Tr_i^M, i.e., $w_i(tr)$, using Lemma 3. Finally, by using Lemma 4, we can easily check whether it is a candidate.

Algorithm 5 Stage 3: Finding candidates

```
 1: procedure MAP($k_1$, $v_1$)
 2:     $Tr_j^R \leftarrow v_1$;
 3:     for $i \leftarrow$1 to $N$ do
 4:         $case \leftarrow$ CHECK($p_i$, $u_i(t)$, $maxU(Tr_i^M)$, $maxU(Tr_j^R)$);
 5:         $k_2 \leftarrow p_i.x\_p_i.y$;
 6:         if $case$=2 then //Partition check
 7:             for $tr \in Tr$ do
 8:                 $v_2 \leftarrow tr$; OUTPUT($k_2$, $v_2$);
 9:         else if $case$=3 then //Trajectory check
10:             for $tr \in Tr$ do
11:                 $subTrList \leftarrow$ SPLIT($tr$, $u_i(t)$);
12:                 for $subTr \in subTrList$ do
13:                     if $subTr$ is a candidate then //Use Lemma 4
14:                         $v_2 \leftarrow tr$; OUTPUT($k_2$, $v_2$);
15: procedure REDUCE($k_2$, $v_2$)
16:     OUTPUT($k_2$, $v_2$);
```

Algorithm 5 gives the detailed pseudocode listings.

Map. The `map()` takes a partition of trajectories Tr_j^R as input. It enumerates each partition Tr_i^M, and checks which case they belong to (lines 3-4). If they belong to case 2, then all the trajectories of Tr_j^R are candidates of Tr_i^M (lines 6-8). Otherwise, for case 3, we split each trajectory of Tr_j^R into a list sub-trajectories according to the breakpoints of $u_i(t)$, and then check them one by one using Lemma 4 (lines 9-14).

Reduce. In the `reduce()`, we simply output the candidates of Tr_i^M, i.e., C_i^R.

We denote the input of `map()` as Tr, i.e., Tr_j^R or a subset of G_j^R. In the `map()`, we first need to enumerate all the partitions and check the cases. If none holds, we need to consider trajectories one by one. Thus, the overall time and space complexities of `map()` are $O(|Tr|Nl)$ and $O(|Tr|l)$ respectively. Since we only need to output the input directly, the time and space complexities of `reduce()` are $O(1)$ and $O(|C_i^R|l)$ respectively.

6.4 Trajectory Join

Since we have found the set, C_i^R, of candidates for each partition Tr_i^M, we join it with C_i^R and then output the k nearest neighbors of each object crossing Tr_i^M. But the result is incomplete since an object may cross several grids. So for each object, we need to reselect k nearest neighbors finally. Algorithm 6 gives the details.

Algorithm 6 Stage 4: Trajectory join

```
1: procedure MAP($k_1$, $v_1$)
2:     $Tr_i^M \leftarrow v_1$;
3:     read $C_i^R$ from Distributed Cache (or HDFS);
4:     $kNNList \leftarrow$ JOIN($Tr_i^M$, $C_i^R$, $k$); // Use BF or SL
5:     for each $tr \in Tr_i^M$ do
6:         OUTPUT($tr.id$, $kNN$); //$k$ nearest neighbors
7: procedure REDUCE($k_2$, $v_2$)
8:     reselect and output $k$ nearest neighbors from $v_2$;
```

Map. The input of `map()` is a partition Tr_i^M (line 2). It joins Tr_i^M with the set C_i^R of corresponding generated candidates, using a single-machine algorithm (lines 3-4). In this paper, we use BF or SL as discussed before, but any other single-machine trajectory join algorithms can be incorporated. Finally, we output a list of pairs (lines 5-6), where the key is the object id and the value is a list of its k nearest neighbors with their minimum distances.

Reduce. The input of `reduce()` is an object with its k nearest neighbors computed from different partitions. We output k objects whose minimum distances are the smallest (lines 8-9).

We denote the input of `map()` as Tr, i.e., Tr_i^M or a subset of G_i^M. Since we need to enumerate each pair of trajectories without extra space cost, the time and space complexities of `map()` are $O(|Tr||C_i^R|l)$. The time and space complexities of `reduce()` are $O(kN)$, since an object may go across at most N grids.

6.5 Enhancing GN with Hashing

We enhance the load balance of the jobs in the workflow of GN, and we call this enhanced approach as GL (Grid with Load balance). Our main idea is to use a hash function to redistribute all the objects in M and R into H disjoint groups respectively. H can be set as a multiple of the maximum number of parallel map tasks running in a cluster (the details of setting H are discussed in the Appendix B). In general, any hash function (e.g., [28]) which can partition objects into groups that keep the same distribution of objects as the overall distribution can be adopted here. In our experiments, since the identifiers of trajectory objects in the dataset are uniformly distributed, we simply hash the objects according to their identifiers, i.e., the hash function is a simple modulo function $hash(tr)$=$tr.id\%H$. After hashing, each group has the same expected number of objects. We denote the trajectories of objects from M (R) in the i (j)-th group as G_i^M (G_j^R). We now discuss the needed modifications for GL.

1. Sub-trajectory extraction. In this stage, all the steps of GL are the same with GN, except we need to redistribute the trajectories using the hash function in the `reduce()`. Specifically, we hash each $tr \in Tr_i^L$ and assign it a key, which is a combination of its set label L and hash value $hash(tr)$. In the output of this job, we output trajectories according to their keys, hence obtaining $2 \times H$ files, corresponding to $G_1^M, \cdots, G_H^M, G_1^R, \cdots, G_H^R$. Note that in each file, trajectories from a same reduce() are collected together and stored in a single line.

2. Computing TDB. In this stage, GL is the same with GN.

3. Finding candidates. In this stage of GL, each map task handles a single group G_j^R, in which each `map()` handles a subset Tr of trajectories from G_j^R, belonging to the same spatial grid.

4. Trajectory join. The principle and implementation of GL is the same as in the finding candidates stage, i.e., each `map()` handles a subset of trajectories from G_i^M.

In the last two stages of GL, each map task handle a group of trajectories, i.e., G_j^R or G_i^M. Since different groups have the same expected size, GL achieves better load balance than GN.

7 The (h,k)-NN Join Algorithm

In this section, we study a variant of the k-NN join, the (h,k)-NN join and the needed adaptations to our framework.

7.1 A Query Algorithm for (h,k)-NN Join

In the (h,k)-NN join, we wish to return h objects of M, having the smallest values of the function f on their k nearest neighbors from R. We call these h objects *target objects*. We assume the aggregate function is *max*, i.e., Equation (4).

A basic solution for (h,k)-NN join is to perform k-NN join firstly and then select h target objects. However, this wastes time with redundant computation, since the k nearest neighbors of each object in M have been retrieved. To perform (h,k)-NN join more efficiently, we propose to compute a new tighter TDB. Recall that Equation (8) gives the TDB of Tr_i^M, which bounds the minimum distance for all the objects, crossing partition Tr_i^M, to their k nearest neighbors. In this equation, the left summand is $maxU(Tr_i^M)$ and the right summand is $v_i(t)$. Now since we only need to return h objects from M, the intuition is to try to make $maxU(Tr_i^M)$ smaller, so that the upper bound will be tighter and thus we achieve higher efficiency in pruning. Figure 12 shows the geometrical intuition.

To adapt the framework of k-NN join for the (h,k)-NN join, we need to perform the following adaptations. In the sub-trajectory extraction stage, for all objects whose trajectories are in Tr_i^M, we select h nearest objects to p_i. In the computing TDB stage, we use the computed statistics on these h objects from each partition to derive a new TDB. Note that the new TDB only ensures that we can find h target objects with their k nearest neighbors, but it cannot ensure that we can return the k nearest neighbors of every object of M correctly. The finding candidates and trajectory join stages are the same with that of k-NN join. Finally, we select and output h target objects with their k nearest neighbors. The size of the results returned by a (h,k)-NN join query is $h \times k$, which is smaller than that of a k-NN join query, i.e., $|M| \times k$.

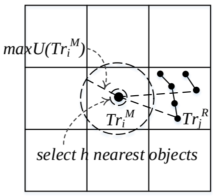


Fig. 12. Main idea

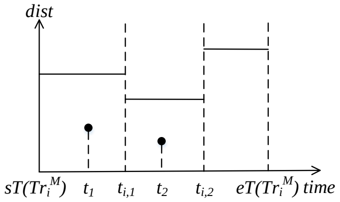


Fig. 13. Computing TDB

7.2 Implementation

1. Sub-trajectory extraction. The `map()` is the same with that of k-NN join. The first steps of `reduce()` are the same with that of k-NN join. Then, if a partition of trajectories is generated by objects from M, i.e., Tr_i^M, we compute $MinDist(tr, p_i)$ for each $tr \in Tr_i^M$, and select h objects whose minimum distances are the smallest. For each of them, we form a triple (id, $dist$, $time$), where id is its identifier, $time$ is the time instance when the minimum distance $dist$ is achieved. Finally, we output these h triples.

2. Computing TDB. The `map()` is the same with that of k-NN join. In the `reduce()`, we first compute the TDB of p_i, i.e., $v_i(t)$, as that in k-NN join. By using triangle inequality, we can easily conclude that the distance from the object whose identifier is id to its k nearest neighbors is at most $dist + v_i(time)$. Thus, for each triple (id, $dist$, $time$) collected in previous stage, we update its $dist$ value to $dist + v_i(time)$. Finally, we output a list of h updated triples in the `reduce()`. Figure 13 gives an example, where $v_i(t)$ is a piece-wise function, and the collected triples are represented by black points at t_1 and t_2.

We collect the output triples of this MapReduce job and select h triples with different ids, having minimal $dist$ values. We denote these h triples as tp_1, tp_2, $\cdots$, tp_h. Then the minimum distance from h target objects to their k nearest neighbors is bounded by $\underset{1 \le j \le h}{max}\, tp_j.dist$. Hence, the TDB of each partition Tr_i^M is:

$$u_i(t) = \underset{1 \le j \le h}{max}\, tp_j.dist, \quad t \in [sT(Tr_i^M), eT(Tr_i^M)]. \quad (17)$$

In the k-NN join case, the maximum distances from all the objects to the central points are used for computing TDB. While in the (h,k)-NN join case, it only uses the minimum distances of h objects from each partition.

3. Finding candidates and trajectory join. Its steps are the same with that of k-NN join.

Finally, for each object of M, we compute the value of the aggregate function on the distances to its k nearest neighbors. Then we output h objects – which have the h smallest values of the aggregate function – with their k nearest neighbors. The time and space complexities of `map()` and `reduce()` in each stage are the same with that of k-NN join.

8 Results

We now present the results. Section 8.1 describes the experiment setup. We then examine k-NN joins and (h,k)-NN joins in Sections 8.2 and 8.3 respectively.

8.1 Setup

Datasets. We validate our approach on synthetic and real datasets:

- **Synthetic data**. We have used two spatial-temporal data simulators: *GSTD* [29] and *Brinkhoff* [30]. For *GSTD*, the moving objects are initially distributed in a $10^4 \times 10^4$ space, and their positions follow a Gaussian distribution with mean 5000 and standard deviation 4000 units. The average speed is 30 units per minute, and the average time between two consecutive points, α, is 1 minute. Two synthetic datasets, **DS1** and **DS2**, are generated, each of which has two sets (M and R) of trajectory objects. In DS1, each set has 10^4 objects, and their positions are monitored for 250 hours; in DS2, each set has 10^6 objects, each of which is observed for 10 hours. The number of points in DS1 (DS2) is 300 million (respectively 1.2 billion). The sizes of DS1 and DS2 are 7GB and 17.2GB respectively. For *Brinkhoff*, we simulate the movement of the objects in the road network of San Joaquin County in California, for 3×10^4 minutes, with α equal to 1 minute. We obtain two sets of trajectory objects, each of which has 6×10^5 objects. We call this dataset **DS3**. The total number of points in DS3 is around 300 million, and its size is 7.3GB.

• **Real data.** We also use the *Beijing taxi* dataset [1], which contains the trajectories of 10,357 taxis in the metropolitan area of Beijing. The locations of the taxis were monitored for a week by on-board GPS devices. Upon removing points that are not in Beijing, a 450MB dataset (of 12.4 million positions) is obtained. The average number of positions associated with each taxi is 1,260. Also, α is 3 minutes on average. We perform a self-join on this dataset, i.e., M and R are the same.

Queries. We investigate the performance of trajectory join queries on the datasets above. By default, $k = 10$. The start time t_s of a query is randomly selected. The default length of the query time interval t_q is selected according to the datasets, as shown in Table 3. We will investigate the effect of k and t_q in Section 8.2.2.

System configuration. We use a MapReduce cluster, which consists of a master node and 60 slave nodes. Each node has a quad-core Intel i7-3770 3.40GHz processor, 16GB of memory, a 1TB hard disk, with Hadoop-2.2.0 installed. All the nodes are connected via Gigabit Ethernet. For the Hadoop configuration, we set the block size of the distributed file system to be 128MB, and the replication factor to be 3. Each node is asked to run 4 map and 4 reduce tasks. Following the official guide[1], we set $60 \times 4 \times 0.95 = 228$ reduce tasks in each MapReduce job. We use the formula, detailed in Appendix B, to obtain the number of hash groups H, which is 240 in our experiments.

We adopt SL as the default single-machine trajectory join algorithm. For the datasets tested, the default number of spatial partitions, N, is 400. The number of temporal partitions, T, is shown in Table 3. Section 8.2.1 studies the effect of these system parameters.

Since the results on our datasets illustrate similar trends, due to the space limitation we only report the most representative results. We mainly focus on the *query execution time*, as well as the *shuffling cost*, which measures the bytes of intermediate results needed to be sent from mappers to reducers.

8.2 Results for k-NN Joins

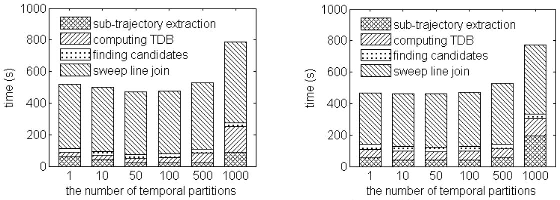
We now report our results. Section 8.2.1 studies the effect of the system parameters. In Section 8.2.2 we investigate the impact of the parameters characterizing the join queries and data.

8.2.1 System parameters

1. Effect of T. Figure 14 shows the performance of the values of T. When T is small, there are few temporal partitions, and thus the time cost of extracting the sub-trajectories in each partition that appears in $[t_s, t_e]$ is high. On the other hand, large values of T lead to more temporal partitions, making the sub-trajectory extraction process more efficient. For example, when T=1, the time cost of extracting sub-trajectories on DS1 is higher than that of T=100. However, larger values of T mean that, the time interval of each temporal partition, Δt=$(T_e - T_s)/T$, can be very small, and thus many new points are inserted when partitioning. This increases the time cost. But the overall performance does not change much within a wide range of T, as shown in the figures. In line with these results, we set T=100 and 10 for these two datasets in the following experiments.

TABLE 3
Default parameter settings

Datasets	System parameters	Query parameters
DS1 (*GSTD*)	T=100, N=400	k=10, t_q=6 hours
DS2 (*GSTD*)	T=10, N=400	k=10, t_q=2 hours
DS3 (*Brinkhoff*)	T=100, N=400	k=10, t_q=10 hours
Beijing taxi	T=10, N=400	k=10, t_q=1 day



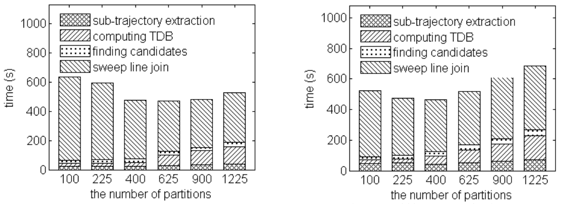
(a) DS1 (b) Beijing taxi data

Fig. 14. Effect of the number of temporal partitions

2. Effect of N. Figure 15 shows how the running time of the algorithms is influenced by the value of N. We also illustrate the breakdown of the running time in each of the four stages. When the value of N increases, the data space is split into more, smaller, grids. Since computing the TDB of a partition is based on the information collected from its nearby partitions, more partitions result in a tighter TDB, which increases pruning power and increases the efficiency. But on the other hand, more partitions increase the number of inserted points lying on the borders of grids, and hence it may create more sub-trajectories. For example, the number of points increases by around 10% when N=400 on DS1. Also, the time cost of computing TDB increases as the value of N increases, since we need to deal with more partitions. Let us take N=1,225 in Figure 15(a) as an example. The running time of computing TDB accounts for 22% of the overall running time. The reason is that, when N=1,225, each grid contains less than 9 objects on average, since $|R|$=10^4. All of them are selected as anchor trajectories since k=10. This demonstrates that computing TDB is costly if we have to consider all the trajectories from R.

We also observe that the efficiency of the solution does not change much for a wide range of values of N. In DS1 dataset, for example, the difference in the efficiency is less than 10% for $N \in [400, 900]$. As long as N is not too small or too large, the performance of the solution is not affected significantly. In our experiments, we set $N = 400$; the size of each partition is 0.25% of the domain space. Among all the 4 stages, the trajectory join stage running time still takes the largest proportion of the time cost, while the sub-trajectory extraction and candidate generation stage take a small proportion of the running time.

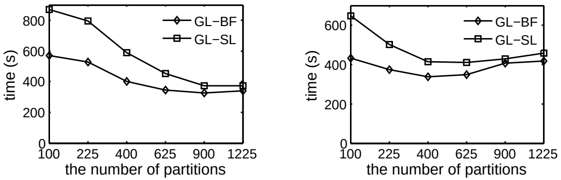
[1] http://wiki.apache.org/hadoop/HowManyMapsAndReduces/



(a) DS1 (b) Beijing taxi data

Fig. 15. Effect of the number of spatial partitions

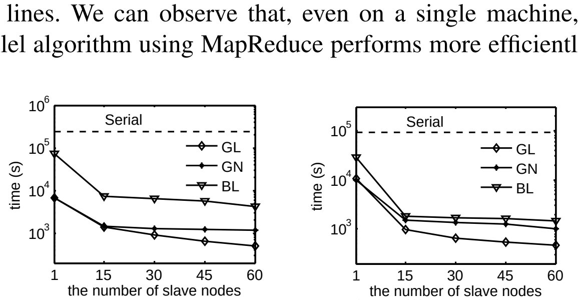
In addition, we compare the efficiency of BF and SL single-machine algorithms, when used in combination with GL. We denote these 2 algorithms as GL-BF and GL-SL respectively. The running time of the trajectory join stage with different values of N is shown in Figure 16. We can observe that GL-SL is consistently more efficient than GL-BF, which indicates that SL is a better choice for our framework. The reason is that, when two trajectories do not have temporal intersection, SL will not consider them as a potential pair and thus performs more efficiently.



(a) DS1 (b) Beijing taxi data

Fig. 16. Comparison between BF and SL

3. Effect of number of nodes. Figure 17 shows the performance of queries with various numbers of computing nodes. As can be observed, the trend is similar on DS1 and Beijing dataset. The running time decreases as the number of nodes increases. Comparing the running time on only 1 slave – equivalent to a single-machine execution – is 22 times slower than the running time on 60 machines. This happens because the number of reduce tasks which run in parallel increases with the number of slave nodes, increasing the computing power. Even on a single machine, we can also see that the running time of GN and GL is still smaller than BL. This shows that our proposed bound is very effective also for pruning on a single machine. When the number of machines increases, the gap in running time between the various algorithms decreases. Moreover, the increase in efficiency displays a slow rate of change. We conjecture this is for two reasons: 1) the shuffling cost increases with the number of machines, and 2) the time cost of the I/O operations on the HDFS increases also. In addition, we also plot the running time of the fully serial algorithm, which runs SL with a single CPU core on a single machine, as shown in the dash lines. We can observe that, even on a single machine, our parallel algorithm using MapReduce performs more efficiently.



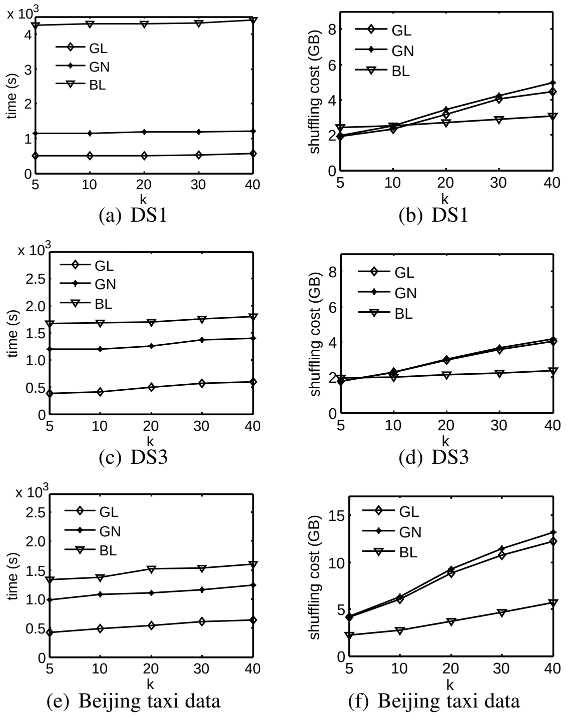
(a) DS1 (b) Beijing taxi data

Fig. 17. Effect of the number of slave nodes

8.2.2 Query and Data Parameters

4. Effect of k. We now investigate how the value of k affects the performance of the k-NN join, in Figure 18. When the value k increases, the running time and shuffling cost of each algorithm increases, because it has to spend more effort on querying more nearest neighbors. For most of the datasets, BL performs slower than GN and GL, which indicates that TDB reduces the computational cost significantly. Moreover, GN performs consistently slower than GL. After partitioning the trajectories using grids, some grids contain more trajectories than the others. In GN, since we join each partition with its candidates in parallel, the running times of different partitions can vary, thus increasing the overall running time. In GL, the distribution of objects into groups according to the hash function enables a better balance, resulting in a better running time. Here, GL performs at least twice faster than GN. In the Beijing taxi dataset, the uniform partitioning of GN results in some grids having more (above 1,000) points than the others. Each partition has 43,000 points on average, but the variance in the number of points in different partitions is 8.56×10^9. In GL, the corresponding variance is less, since a hash function is adopted to redistribute the trajectories.

We observed that the shuffling cost of GN and GL is larger than that of BL. Most of the shuffling cost is invested on TDB computation, where for each anchor trajectory, GN and GL need to transmit the maximum distance values to all the central points. On the other hand, the use of TDB reduces the amount of data computation effectively. Thus, the overall running time of GL and GN is better than BL. Notice that when the data size increases, the shuffling cost of GL and GN can be smaller than that of BL, as we will discuss later in the scalability experiment.



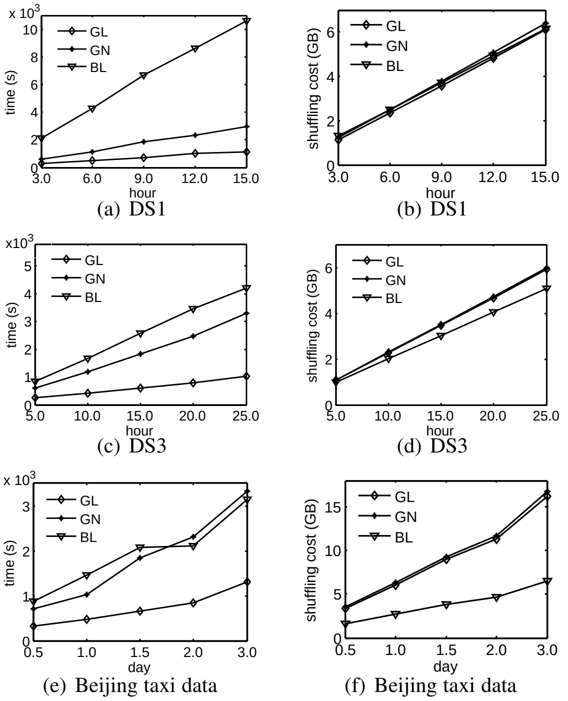
(a) DS1 (b) DS1

(c) DS3 (d) DS3

(e) Beijing taxi data (f) Beijing taxi data

Fig. 18. Effect of k

5. Effect of t_q. Figure 19 shows the performance of queries with different t_q on three datasets. A noticeable effect is the linear dependence of the running time on t_q, as an increase in t_q usually involves more points on the trajectory to process. These results are similar to what we have observed in the case of varying k, i.e., GL performs consistently faster than BL and GN and that the TDB pruning and load balancing are highly effective.



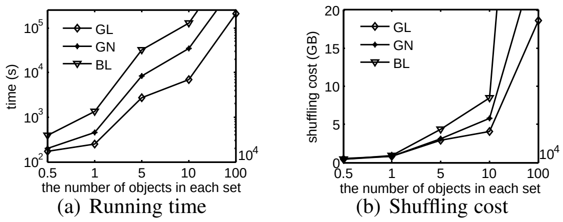
(a) DS1 (b) DS1

(c) DS3 (d) DS3

(e) Beijing taxi data (f) Beijing taxi data

Fig. 19. Effect of the length of query time interval

6. Scalability. Figure 20 shows the scalability of algorithms with respect to M and R (When $|M|$=$|R|$=10^6, we stop running GN and BL after 3 days). The execution and shuffling costs increase with $|M|$ and $|R|$, because more objects are involved in the join. Notice that the execution and shuffling costs of GN and GL grow slower than that of BL. When $|M|$=$|R|$=10^4, BL is 6 times slower than GL, while the shuffling cost remains roughly the same. When $|M|$=$|R|$=10^5, the running time of BL is 18 times larger than that of GL, while the shuffling cost increases to 2 times over GL. To explain this, notice that when data size increases, the number of anchor trajectories becomes a smaller fraction of the trajectories generated by objects in R. This is because k is often smaller than $|R|$. Hence, the shuffling costs of GL and GN become smaller than those of the basic solution BL.

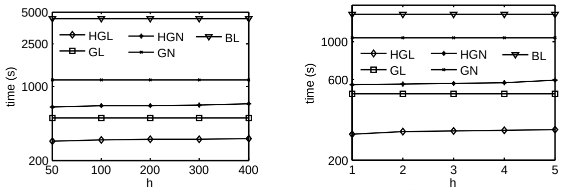


(a) Running time (b) Shuffling cost

Fig. 20. Scalability on DS2

8.3 Results for (h, k)-NN Joins

Finally, we examine our modified algorithms for GN and GL, namely HGN and HGL, for evaluating the (h, k)-NN join (Section 7). We compare them with a baseline solution, which runs the k-NN join, evaluates f on objects in M, and then selects h ones with their k nearest neighbors. Figure 21 shows the results under different h values, with k set to 10. BL is the slowest, since it does not do any pruning. HGL and HGN run about twice faster than GL and GN respectively. For example, when h=50, HGN and GN costs 550 and 1050 seconds respectively. The reason is that the new TDB method designed for the (h, k)-NN join is tighter than the TDB designed only for the k-NN join, thereby reducing the computational cost significantly.



(a) DS1 (b) Beijing taxi data

Fig. 21. Effect of h on (h, k)-NN join

9 Conclusions

We study the k-NN join for big trajectory data. We present a efficient solution for answering k-NN join queries based on MapReduce. We also develop a fast pruning technique for the (h, k)-NN join. In the future, we will examine the evaluation of other queries for trajectories. We will also study how our solution can be executed in other parallel platforms.

Acknowledgment

Reynold Cheng and Yixiang Fang were supported by RGC (Project HKU 17205115). We thank the reviewers for their comments.

References

[1] Y. Jing et al, “T-drive: Driving directions based on taxi trajectories,” *Association for Computing Machinery*, nov. 2010.

[2] W. Luo, H. Tan, L. Chen, and L. M. Ni, “Finding time period-based most frequent path in big trajectory data,” in *SIGMOD*, 2013, pp. 713–724.

[3] Y. Zheng and X. Zhou, *Computing with Spatial Trajectories*. Springer, November 2011.

[4] S. Arumugam and C. Jermaine, “Closest-point-of-approach join for moving object histories.” in *ICDE*, 2006, p. 86.

[5] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, “Nearest neighbor search on moving object trajectories,” in *SSTD*, 2005.

[6] Y. Chen and J. M. Patel, “Design and evaluation of trajectory join algorithms.” in *GIS*, 2009.

[7] N. A. Bahcall et al, “The spatial correlation function of rich clusters of galaxies,” *The Astrophysical Journal*, vol. 270, pp. 20–38, 1983.

[8] R. Benetis et al, “Nearest and reverse nearest neighbor queries for moving objects,” *The VLDB Journal*, vol. 15, no. 3, 2006.

[9] http://hubblesite.org/the_telescope/hubble_essentials/quick_facts.php.

[10] http://hubblesite.org/newscenter/archive/releases/2013/22/full/.

[11] J. Duggan et al, “Skew-aware join optimization for array databases,” in *SIGMOD*, 2015, pp. 123–135.

[12] T. J. Henry et al, “The solar neighborhood iv: discovery of the twentieth nearest star,” *The Astronomical Journal*, vol. 114, pp. 388–395, 1997.

[13] J. Kartaltepe et al, “Evolution of the frequency of luminous close galaxy pairs at z¡ 1.2 in the cosmos field,” *The Astrophysical Journal Supplement Series*, vol. 172, no. 1, p. 320, 2007.

[14] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *OSDI*, 2004, pp. 10–10.

[15] A. Akdogan et al, “Voronoi-based geospatial query processing with mapreduce.” in *CloudCom*, 2010, pp. 9–16.

[16] W. Zhang, R. Cheng, and B. Kao, “Evaluating multi-way joins over discounted hitting time.” in *ICDE*, 2014, pp. 724–735.

[17] Y. Tao, D. Papadias, and Q. Shen, “Continuous nearest neighbor search,” in *VLDB*. VLDB Endowment, 2002, pp. 287–298.

[18] M. F. Mokbel et al, “Sina: Scalable incremental processing of continuous queries in spatio-temporal databases,” in *SIGMOD*, 2004.

[19] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *VLDB*, 1998, pp. 570–581.

[20] J. Kearney and S. Hansen, "Stream editing for animation," in *Technical report, DTIC Document*, 1990.

[21] L. Chen, M. T. Özsu, and V. Oria, "Robust and fast similarity search for moving object trajectories," in *SIGMOD*, 2005, pp. 491–502.

[22] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *SIGMOD*, 2013, pp. 13–24.

[23] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *SIGMOD*, 2010, pp. 495–506.

[24] X. Zhang et al, "Efficient multi-way theta-join processing using mapreduce," *VLDB Endowment*, vol. 5, no. 11, pp. 1184–1195, 2012.

[25] C. Zhang, F. Li, and J. Jestes, "Efficient parallel knn joins for large data in mapreduce," in *EDBT*, 2012, pp. 38–49.

[26] W. Lu et al, "Efficient processing of k nearest neighbor joins using mapreduce," *Proc. VLDB Endow.*, vol. 5, no. 10, pp. 1016–1027, 2012.

[27] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[28] M.-W. Lee and S.-w. Hwang, "Robust distributed indexing for locality-skewed workloads," in *CIKM*, 2012, pp. 1342–1351.

[29] Y. Theodoridis et al, "On the generation of spatiotemporal datasets," *Advances in Spatial Databases*, pp. 147–164, Sep. 1999.

[30] T. Brinkhoff, "A framework for generating network-based moving objects," *Geoinformatica*, vol. 6, no. 2, pp. 153–180, Jun. 2002.

Appendix A
Anchor trajectory selection

Algorithm 7 Selecting anchor trajectories

```
1:  procedure SELECTANCHOR($Tr_j^R$, $startT$, $k$)
2:      init $Q$, $achList$;
3:      while $Q$.size $< k$ do
4:          $tr \leftarrow$ FINDNEXT($Tr_j^R$, $startT$);
5:          $Q$.add($tr$);
6:      while $Q$.size $> 0$ do
7:          $achTr \leftarrow Q$.pop; $achList$.add($achTr$);
8:          $startT \leftarrow achTr.e$;
9:          $tr \leftarrow$ FINDNEXT($Tr_j^R$, $startT$);
10:         if $tr$!=$null$ then $Q$.add($tr$);
        return $achList$;
11: procedure FINDNEXT($Tr_j^R$, $startT$)
12:     $minT \leftarrow \infty$, $achTr \leftarrow null$;
13:     for $tr \in Tr_j^R$ do
14:         $t' \leftarrow tr.s - startT$;
15:         if $t' < 0$ then $t' \leftarrow 0$;
16:         if $t' < minT$ then
17:             $achTr \leftarrow tr$, $minT \leftarrow t'$;
18:         else if $t' = minT$ then
19:             $d \leftarrow MaxDist(achTr, p_j)$, $d' \leftarrow MaxDist(tr, p_j)$;
20:             if $d < d'$ then $achTr \leftarrow tr$;
21:     if $achTr$!=$null$ then
22:         $Tr_j^R$.remove($achTr$);
23:         $achTr \leftarrow achTr$.subTraj($startT + minT$, $achTr.e$);
        return $achTr$;
```

We propose a heuristic algorithm to select the anchor trajectories as shown in Algorithm 7. It initializes a priority queue Q (line 2), in which trajectories are sorted based on the end time in ascending order. k anchor trajectories are selected by calling FINDNEXT($startT$) (lines 3-5). Then we dequeue an anchor trajectory from Q and update $startT$ (lines 7-8). A new anchor trajectory is selected and enqueued into Q (lines 9-10). This process (lines 7-10) is iterated until all the anchor trajectories are dequeued. In FINDNEXT($startT$), for each tr, we compute the gap between its start time and $startT$ (lines 14-15), and select the trajectory whose start time is closest to $startT$ (lines 16-17). If more than one trajectory having this minimal gap, we select the one whose maximum distance to the central point is the smallest (lines 18-20). Finally, we get this anchor trajectory (lines 21-23).

Appendix B
Computing the Value of H

Suppose the block size of HDFS is S, the storage cost for a point in a trajectory is s, the maximum number of map tasks that can be run in parallel in a cluster as Mp, the average number of points in a unit time interval is n. For a specific k-NN join query with time interval length t_q, the expected total storage space for points generated by objects from M, which should be involved in the query processing, is $|M| \times t_q \times n \times s$. So the minimum number of blocks to store these data in HDFS is $\frac{|M| \times t_q \times n \times s}{S}$.

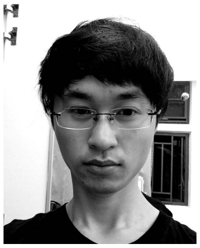
In a MapReduce job, the number of map tasks equals to the number of blocks of the input data. Since only Mp map tasks can be run in parallel, to achieve good load balance for handling set M, the number of groups for hashing, i.e., Tr^M, can be set as:

$$H = \left\lceil \frac{|M| \times t_q \times n \times s}{S \times Mp} \right\rceil \times Mp. \tag{18}$$

Similarly, we can compute H^R for points generated by objects from R. For simplicity, in this paper, we set H=$\max\{H^M, H^R\}$.

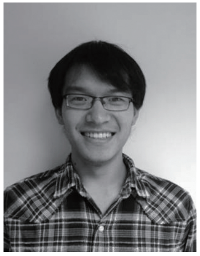
Yixiang Fang received the B.Eng. and M.S. degrees from Harbin Engineering University and ShenZhen Graduate School in Harbin Institute of Technology in 2010 and 2013, respectively. He is currently working towards the Ph.D. degree in Department of Computer Science in the University of Hong Kong (HKU) under the supervision of Dr. Reynold Cheng. His research interests mainly focus on big data analytics on spatial-temporal databases, graph databases, uncertain databases, and Web data mining.

Reynold Cheng is an Associate Professor in the Department of Computer Science of the University of Hong Kong (HKU). He obtained his PhD from Department of Computer Science of Purdue University in 2005. Dr. Cheng was granted an Outstanding Young Researcher Award 2011-12 by HKU. He has served as PC members and reviewers for international conferences (e.g., SIGMOD, VLDB, ICDE, EDBT, KDD and ICDM) and journals (e.g., TKDE, TODS, VLDBJ and IS).



Wenbin Tang received the B.Eng. degree in computer science and technology from Fuzhou University in 2012. He is now a M.Phil. student in the University of Hong Kong (HKU) under the supervision of Dr. Hubert Chan. His research interests include algorithms and artificial intelligence.

Silviu Maniu is an Associate Professor at Université Paris-Sud in Orsay, France. Before, he was a Researcher in Huawei Noah's Ark Lab in Hong Kong, and a Postdoctoral Fellow at the University of Hong Kong. He received the Ph.D. degree in computer science from Télécom ParisTech in Paris, France in 2012. His research interests include graph databases, uncertain data management and social data management.



Xuan Yang received the B.Sc. degree in computer science from Fudan University in 2009. In 2014, he received the Ph.D. degree from the Department of Computer Science in the University of Hong Kong under the supervision of Dr. Reynold Cheng and Prof. David Cheung. His research interests include uncertain data management, data cleaning and web data mining.